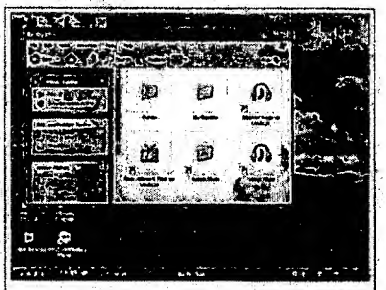
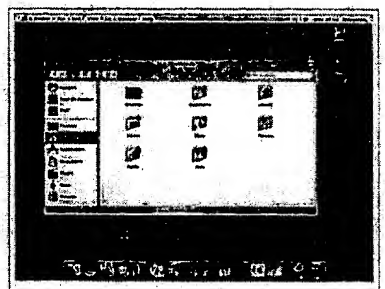
Graphical user interface

From Wikipedia, the free encyclopedia.

A **graphical user interface** (or **GUI**, pronounced "gooey") is a method of interacting with a computer through a metaphor of direct manipulation of graphical images and widgets in addition to text.



An example of graphical user interface in Microsoft Windows



An example of graphical user interface in Mac OS X

Contents

GUIs and PUIs

The precursor to GUIs was invented by researchers at the Stanford Research Institute (led by Doug Engelbart) with the development and use of text-based hyperlinks manipulated with a mouse for the On-Line System. The concept of hyperlinks was further refined and extended to graphics by researchers at Xerox PARC, who went beyond text-based hyperlinks and used GUIs as the primary interface for the Xerox Alto computer. Most modern general-purpose GUIs are derived from this system. For this reason some people call this class of interface a *PARC User Interface* (PUI) (note that PUI is also an acronym for *perceptual user interface*). The PUI consists of graphical widgets (often provided by widget toolkit libraries) such as windows, menus, radio buttons, check boxes, and icons, and employs a pointing device (such as a mouse, trackball, or touchscreen) in addition to a keyboard. For this reason, many supporters of command Line Interface operating systems once referred to PUIs as WIMPs, which stood for Windows, Icons, Menus, Pointer. The term *GUI* is used to describe the user interface of most modern operating systems, although occasionally other metaphors surface, such as Microsoft Bob, 3dwm or (partially) FSV.

Examples of systems that support GUIs are Mac OS, Microsoft Windows, NEXTSTEP and the X Window System. The latter is extended with toolkits such as Motif (CDE), Qt (KDE) and GTK+ (GNOME).

Types of GUIs

GUIs that are not PUIs are most notably found in computer games, and advanced GUIs based on virtual reality

are now frequently found in research. Many research groups in North America and Europe are currently working on the Zooming User Interface or ZUI, which is a logical advancement on the GUI, blending some 3D movement with 2D or "2 and a half D" vectorial objects.

Some GUIs are designed for the rigorous requirements of vertical markets. These are known as "application specific GUIs." One example of such an application specific GUI is the now familiar touchscreen point of sale software found in restaurants worldwide and being introduced into self-service retail checkouts. First pioneered by Gene Mosher on the Atari ST computer in 1986, the application specific touchscreen GUI has spearheaded a worldwide revolution in the use of computers throughout the food & beverage industry and in general retail.

Other examples of application specific touchscreen GUIs include the most recent automatic teller machines, airline self-ticketing, information kiosks and the monitor/control screens in embedded industrial applications which employ a real time operating system (RTOS). The latest cell phones and handheld game systems also employ application specific touchscreen GUIs.

GUIs versus CLIs

The GUI is generally contrasted with the command line interface (CLI), an earlier, text-only interface that required the user to type in commands or text strings to cause the computer to take some action. Between these two types but more similar to GUIs are text user interfaces (TUIs) that display the same types of widgets as a GUI but in a character-cell text mode rather than in a pixel-based graphics mode. Examples include the interfaces of many ncurses and DOS applications.

Because GUIs and TUIs tend to show most or all relevant categories of commands on the display, users often learn them faster than CLIs. However, since the choice of displayed options to choose from has been made for the user and is usually more limited than the full set of options available, full use of all the software's functionality on a GUI system often takes considerable time. By contrast, a CLI typically makes all options and choices equally accessible but also equally invisible and not easily remembered, and so mastering a CLI generally requires more extensive familiarity with the software's features and functionality. A somewhat caustic comment about the pre-OS X Macintosh interface captures this: "you can learn to use a Macintosh in 30 minutes, but after six months you will have learned nothing more about using a Macintosh."

Users with vision or motion disabilities often have trouble navigating in a GUI, and most commercial GUIs require at least an order of magnitude more computer power (CPU speed, RAM, disk space, display resolution and response, etc.) than a CLI, making a GUI unwieldy on less expensive, smaller, or older hardware. Designing suitable interfaces for handheld devices, such as PDA applications and their smartphone cousins, has been a major challenge for user interface designers, and some of the more successful diverge considerably from desktop computer designs.

The nature of GUIs

A certain amount of insight into GUIs can be obtained by comparing noun-verb to verb-noun metaphors. Noun-verb interaction begins by picking an object then telling the system what to do to it. Verb-noun systems tell the system what to do, then pick the object to do it to. Most GUIs are implemented in terms of an event model, although other models exist. These alternative models for creating GUIs are generally classed as user interface management systems or UIMS.

In academic and research circles a GUI is often referred to as a Direct manipulation interface. This term was coined and adopted in the late 1980s because it was felt the term "Graphic User Interface" did not reflect the actual physical or haptic reality of manipulating a mouse or using a touch screen and that it ignored completely

the coordinated use of sound effects to support the manipulation of the graphic elements in this kind of user interface. Also, academic and research institutions often work on prototypes of future user interfaces that place an equal or greater emphasis on the tactile elements of the interface. The "direct manipulation interface" term is usually not presented as an acronym.

See also

- History of the GUI
- GUI toolkit
- UIML
- Fitts' law
- Anti-Mac
- Apple v. Microsoft
- User interface engineering
- Software engineering
- List of software engineering topics
- Human-Machine Interface
- ergonomics
- GUI Testing

External links

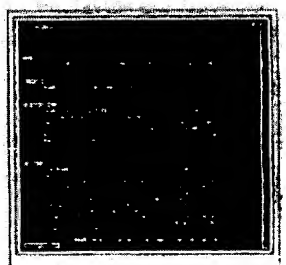
- Marcin Wichary's GUIdebook (http://www.aci.com.pl/mwichary/guidebook/), Graphical User Interface gallery: over 2800 screenshots of UI history
- The Real History of the GUI (http://www.sitepoint.com/article/real-history-gui), a very interesting article by Mike Tuck
- A History of the GUI (http://arstechnica.com/articles/paedia/gui.ars), by Jeremy Reimer of Ars Technica

Retrieved from "http://en.wikipedia.org/wiki/Graphical_user_interface"

Categories: Human-computer interaction | Graphical user interface | Software architecture

Command line interface

From Wikipedia, the free encyclopedia.



Rxvt is a VT102 terminal emulator

A **command line interface** or **CLI** is a method of interacting with a computer by giving it lines of textual commands (that is, a sequence of characters) either from keyboard input or from a script. It is occasionally also referred to as a **CLUE**, for Command Line User Environment. In its simplest form, the user types a command after the computer displays a prompt. The computer then carries out the command given. The term is usually used in contrast to a graphical user interface (GUI) in which commands are typically issued by moving a pointer (via a pointing device) and/or pressing a key (that is, by "clicking", often on a key mounted on a mouse).

Programs that implement these interfaces are often called command line interpreters. Examples of such programs include the various different Unix shells, VMS' DCL (Digital Command Language), and related designs like CP/M and DOS's command.com, both based heavily on DEC's RSX and RSTS operating system interfaces (which were also command line interfaces). Microsoft claims their next major operating system, code-named Longhorn, will include an enhanced command line interface named MSH (Microsoft Shell, codename *Monad*), which combines the features of traditional Unix shells with the object-oriented .NET framework.

Some applications provide command lines as well. The CAD program AutoCAD is a prominent example. In some computing environments like the Oberon or Smalltalk user interface, most of the text which appears on the screen may be used for giving commands.

The commands given on a command line interface are often of the form

```
[doSomething] [how] [toAFile]
```

or

```
[doSomething] [how] < [inputFile] > [outputFile]
```

doSomething corresponds to a verb, *how* to an adverb (it describes how the command should be performed in this instance—for example, should it be particularly "verbose", or particularly "quiet") and *toAFile* to an object (often one or more files) against which the command should be run. The standalone '>' in the second example is a redirection character, telling the operating system (i.e., usually a command shell interpreter) to send the output of the previous commands (that is, those on the left of '>') to some other place (that is, the file named to the right of the '>'). Another common and important redirection character is the pipe ('|'), which tells the CLI to treat the output of this command as the input of another; this can be a very powerful mechanism for the user, as explained under pipeline (Unix) and pipes and filters.

Contents

Advantages of a command line interface

Even though new users seem to learn GUIs more quickly to perform common operations, well-designed CLIs have several advantages:

- Skilled users are able to use a command line faster than a GUI for many tasks. This advantage is magnified by tab completion and by the fact that programs intended to be run from the command line are often given very short names.
- Options and operations are often invokable in a consistent form, one "level" away from the basic command. With most GUIs, the available operations and options may appear on different menus with differing usage patterns. They may be separated on several different menu levels as well. In either case, different applications (or utilities) may have different patterns; if so there is little advantage in either approach. Both are likely to annoy users.
- All options and operations are controlled in more or less the same way. The "more or less" in this case is a common accusation against CLIs: it *should* be no more difficult to understand and perform a rare operation than a common one, but in practice it may require learning previously unencountered syntax. However, few GUIs offer even comparable access to their entire range of available options.
- CLIs can often double as scripting programming languages and can perform operations in a batch processing mode without user interaction. That means that once an operation is analyzed and understood, a "script" implementing that understanding can be written and saved. The operation can thereafter be carried out with no further analysis and design effort. With GUIs, users must start over at the beginning every time, as GUI scripting (if available at all) is almost always more limited—although macros can sometimes be used in a similar way. Simple commands do not even need an actual script, as the completed command can usually be assigned a name (an "alias") and executed simply by typing that name into the CLI.

Disavantages of a command line interface

- CLIs have a steeper learning curve than GUIs—more time and effort must be devoted to learning the basics before even simple tasks may be accomplished.
- CLIs are unsuited to certain tasks, such as image and sound editing.

See also

- Text user interface

External links

- "In the Beginning was the Command Line" (http://www.spack.org/index.cgi/InTheBeginningWasTheCommandLine) By Neal Stephenson - A gentle introduction to the difference between command line interfaces and GUIs, and the history of their development.
- The Interaction-Design.org Encyclopedia entry on Interaction Styles, comparing Command Line Interfaces with other Interaction Styles (http://www.interaction-design.org/encyclopedia/interaction_styles.html)

- *Coming Soon to Windows: The Microsoft Shell (MSH)* by Jason Nadal (http://www.developer.com/net/net/article.php/3286851)

Retrieved from "http://en.wikipedia.org/wiki/Command_line_interface"

Categories: Human-computer interaction | Computing | Software engineering | Software architecture

---

Unix shell

From Wikipedia, the free encyclopedia.

A **Unix shell**, also called "the command line", provides the traditional user interface for the Unix operating system. Users direct the operation of the computer by entering command input as text for a shell to execute. Within the Microsoft Windows suite of operating systems the analogous program is command.com, or cmd.exe for Windows NT-based operating systems.

The most generic sense of the term *shell* means *any* program that users use to type commands; it is called a "shell" because it hides the details of the underlying operating system behind the shell's interface (contrast "kernel", which refers to the lowest-level, or 'inner-most' component of an operating system). Similarly, graphical user interfaces for Unix, such as GNOME and KDE, are sometimes called *visual shells* or *graphical shells*. By itself, the term *shell* is usually associated with the command line. In Unix, any program can be the user's shell; users who want to use a different syntax for typing commands can specify a different program as their shell.

The term *shell* also refers to a particular program, namely the Bourne shell, `sh`. The Bourne shell was the shell used in early versions of Unix and became a *de facto* standard; every Unix-like system has the equivalent of the Bourne shell. The Bourne shell program is located in the UNIX file hierarchy at `/bin/sh`. On some systems, such as BSD, `/bin/sh` is a Bourne shell or equivalent, but on other systems such as Linux, `/bin/sh` is likely to be a link to a compatible, but more feature-rich shell, such as Bash. POSIX specifies the standard shell as a strict subset of the Korn shell.

The Unix shell is unusual since it is in both an interactive command language and the language used to script the system; it is a scripting programming language.

On systems using a windowing system, some users may never use the shell directly, though on Unix systems, the shell is still the implementation language of system startup scripts, including the program that starts the windowing system, the program that dials into the Internet, and many other essential functions.

On Windows, equivalents to Unix system scripts are called batch files, and have a ".bat" extension.

Many regular users of a UNIX system still find a modern command line shell much more convenient for many tasks than any GUI application.

Contents

Unix shells

- Almquist shell (ash)
- Bourne-Again shell (bash)
- Bourne shell (sh) Written by Steve Bourne, while at Bell Labs. First distributed with Version 7 UNIX, circa 1978.
- C shell (csh) Written by Bill Joy, while at the University of California, Berkeley. First distributed with BSD, circa 1979.

- es shell (es) A functional programming rc-compatible shell written in the mid-1990s.
- fish, first released in 2005.
- Korn shell (ksh) Written by David Korn, while at Bell Labs.
- rc shell (rc) The Plan 9 shell by Tom Duff while at Bell Labs, later backported to Unix and other Operating Systems.
- scsh (Scheme Shell)
- TENEX C shell (tcsh)
- Z shell (zsh)

Historic

- Thompson shell (sh) The first Unix shell, written by Ken Thompson at Bell Labs. Distributed with Versions 1 through 6 of Unix, from 1971 to 1975.
- PWB shell (sh) A version of the Thompson shell, augmented by John Mashey and others, while at Bell Labs. Distributed with the Programmer's Workbench UNIX, circa 1976.

See also

- Shebang
- Shell script
- List of Unix programs
- Shell account

External links

- *Unix Shells - csh, ksh, bash, zsh, ...* by Christopher Browne (http://cbbrowne.com/info/unixshells.html)

Retrieved from "http://en.wikipedia.org/wiki/Unix_shell"

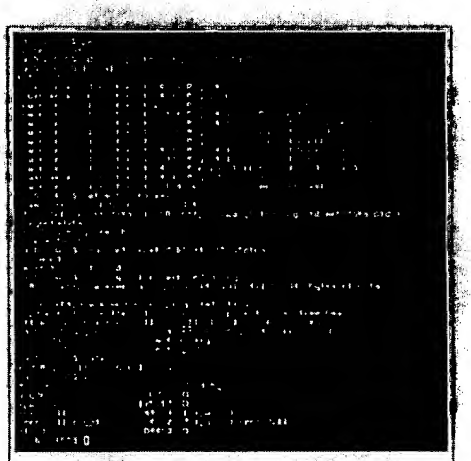
Categories: Unix programs | System administration

Bash

From Wikipedia, the free encyclopedia.
(Redirected from Bourne-Again shell)

This article is about the UNIX shell named Bash. For information about the popular website with humorous quotes, see Bash.org.

Bash is a UNIX command shell written for the GNU project. Its name is an acronym for *Bourne-again shell* — a pun on the Bourne shell (sh), which was an early, important UNIX shell. The Bourne shell was the shell distributed with Version 7 UNIX, circa 1978. The original Bourne shell was written by Stephen Bourne, then a researcher at Bell Labs. The Bash shell was written in 1987 by Brian Fox. In 1990, Chet Ramey became the primary maintainer. Bash is the default shell on most GNU/Linux systems as well as on Mac OS X Tiger, and it can be run on most UNIX-like operating systems. It has also been ported to Microsoft Windows by the Cygwin project.



Screenshot of a sample Bash session, taken on Linux.

Contents

Bash syntax highlights

Bash's command syntax is a superset of the Bourne shell's command syntax. The definitive specification of Bash's command syntax is the man page that's installed along with Bash. This section highlights some of Bash's unique syntax features.

The vast majority of Bourne shell scripts can be executed without alteration by Bash, with the exception of those Bourne shell scripts that happen to reference a Bash special variable or to use a Bash builtin command. The Bash command syntax includes ideas drawn from the Korn shell (ksh) and the C shell (csh), such as command-line editing, command history, the directory stack, the `$RANDOM` and `$PPID` variables, and POSIX command substitution syntax: `$(...)`. When being used as an interactive command shell, Bash supports completion of partly typed-in program names, filenames, variable names, etc. when the user presses the TAB key.

Bash syntax has many extensions that the Bourne shell lacks. Several of those extensions are enumerated here.

Integer mathematics

A major limitation of the Bourne shell is that it cannot perform integer calculations without spawning an external process. Bash can perform in-process integer calculations using the `(( ... ))` command and the `$[ ... ]` variable syntax, as follows:

```
VAR=55              # Assign integer 55 to variable VAR.
((VAR = VAR + 1))   # Add one to variable VAR.  Note the absence of the '$' character.
((++VAR))           # Another way to add one to VAR.  Performs C-style pre-increment.
((VAR++))           # Another way to add one to VAR.  Performs C-style post-increment.
echo $[VAR * 22]    # Multiply VAR by 22 and substitute the result into the command.
echo $((VAR * 22)) # Another way to do the above.
```

The `(( ... ))` command can also be used in conditional statements, because its exit status is 0 or 1 depending on whether the condition is true or false:

```
if ((VAR == Y * 3 + X * 2))
then
        echo Yes
fi

((Z > 23)) && echo Yes
```

The `(( ... ))` command supports the following relational operators: '==', '!=', '>', '<', '>=', and '<='.

Bash cannot perform in-process floating point calculations. The only UNIX command shells capable of this are Korn Shell (1993 version) and zsh (starting at version 4.0).

I/O redirection

Bash has several I/O redirection syntaxes that the traditional Bourne shell lacks. Bash can redirect standard output and standard error at the same time using this syntax:

```
command &> file
```

which is simpler to type than the equivalent Bourne shell syntax, "`command > file 2>&1`". Bash can redirect standard input from a string using the following syntax:

```
command <<< "string to be read as standard input"
```

If the string contains whitespace, it must be quoted.

Example: Redirect standard output to a file, write data, close file, reset stdout

```
# make Filedescriptor(FD) 6 a copy of stdout (FD 1)
exec 6>&1
# open file "test.data" for writing
exec 1>test.data
# produce some content
echo "data:data:data"
# close file "test.data"
exec 1>&-
# make stdout a copy of FD 6 (reset stdout)
```

```
exec 1>&6
# close FD6
exec 6>&-
```

Open and close files

```
# open file test.data for reading
exec 6<test.data
# read until end of file
while read -u 6 dta
do
  echo "$dta"
done
# close file test.data
exec 6<&-
```

Catch output of external commands

```
# execute 'find' and store results in VAR
# search for filenames which end with the letter "h"
VAR=$(find . -name "*h")
```

In-process regular expressions

Bash 3.0 supports in-process regular expression matching using the following syntax, reminiscent of Perl:

```
[[ string =~ regex ]]
```

The regular expression syntax is the same as that documented by the regex(3) man page. The exit status of the above command is 0 if the regex matches the string, 1 if it does not match. Parenthesized subexpressions in the regular expression can be accessed using the shell variable `BASH_REMATCH`, as follows:

```
if ([ abcfoobarbletch =~ 'foo(bar)bl(.*)' ]]
then
        echo The regex matches!
        echo $BASH_REMATCH      -- outputs: foobarbletch
        echo ${BASH_REMATCH[1]} -- outputs: bar
        echo ${BASH_REMATCH[2]} -- outputs: etch
fi
```

This syntax gives performance superior to spawning a separate process to run a `grep` command, because the regular expression matching takes place within the Bash process. If the regular expression or the string contain whitespace or shell metacharacters (such as '*' or '?'), they should be quoted.

Backslash escapes

Words of the form `$'string'` are treated specially. The word expands to `string`, with backslash-escaped characters replaced as specified by the C programming language. Backslash escape sequences, if present, are decoded as follows:

Backslash Escapes

Backslash Escape	Expands To ...
\a	An alert (bell) character
\b	A backspace character
\e	An escape character
\f	A form feed character
\n	A new line character
\r	A carriage return character
\t	A horizontal tab character
\v	A vertical tab character
\\	A backslash character
\'	A single quote character
\nnn	The eight-bit character whose value is the octal value nnn (one to three digits)
\xHH	The eight-bit character whose value is the hexadecimal value HH (one or two hex digits)
\cx	A control-X character

The expanded result is single-quoted, as if the dollar sign had not been present.

A double-quoted string preceded by a dollar sign (`$"..."`) will cause the string to be translated according to the current locale. If the current locale is C or POSIX, the dollar sign is ignored. If the string is translated and replaced, the replacement is double-quoted.

Bash startup scripts

When Bash starts, it executes the commands in a variety of different scripts.

When Bash is invoked as an interactive login shell, or as a non-interactive shell with the `--login` option, it first reads and executes commands from the file `/etc/profile`, if that file exists. After reading that file, it looks for `~/.bash_profile`, `~/.bash_login`, and `~/.profile`, in that order, and reads and executes commands from the first one that exists and is readable. The `--noprofile` option may be used when the shell is started to inhibit this behavior.

When a login shell exits, Bash reads and executes commands from the file `~/.bash_logout`, if it exists.

When an interactive shell that is not a login shell is started, Bash reads and executes commands from `~/.bashrc`, if that file exists. This may be inhibited by using the `--norc` option. The `--rcfile file` option will force Bash to read and execute commands from `file` instead of `~/.bashrc`.

When Bash is started non-interactively, to run a shell script, for example, it looks for the variable BASH_ENV in the environment, expands its value if it appears there, and uses the expanded value as the name of a file to read

and execute. Bash behaves as if the following command were executed:

```
if [ -n "$BASH_ENV" ]; then . "$BASH_ENV"; fi
```

but the value of the PATH variable is not used to search for the file name.

If Bash is invoked with the name sh, it tries to mimic the startup behavior of historical versions of sh as closely as possible, while conforming to the POSIX standard as well. When invoked as an interactive login shell, or a non-interactive shell with the `--login` option, it first attempts to read and execute commands from `/etc/profile` and `~/.profile`, in that order. The `--noprofile` option may be used to inhibit this behavior. When invoked as an interactive shell with the name sh, Bash looks for the variable ENV, expands its value if it is defined, and uses the expanded value as the name of a file to read and execute. Since a shell invoked as sh does not attempt to read and execute commands from any other startup files, the `--rcfile` option has no effect. A non-interactive shell invoked with the name sh does not attempt to read any other startup files. When invoked as sh, Bash enters *posix* mode after the startup files are read.

When Bash is started in posix mode, as with the `--posix` command line option, it follows the POSIX standard for startup files. In this mode, interactive shells expand the ENV variable and commands are read and executed from the file whose name is the expanded value. No other startup files are read.

Bash attempts to determine when it is being run by the remote shell daemon, usually `rshd`. If Bash determines it is being run by `rshd`, it reads and executes commands from `~/.bashrc`, if that file exists and is readable. It will not do this if invoked as sh. The `--norc` option may be used to inhibit this behavior, and the `--rcfile` option may be used to force another file to be read, but `rshd` does not generally invoke the shell with those options or allow them to be specified.

External links

- Bash home page (http://www.gnu.org/software/bash/)
- Bash FAQ (ftp://ftp.cwru.edu/pub/bash/FAQ)
- Bash 3.0 Announcement (http://groups.google.com/groups?dq=&lr=&ie=UTF-8&group=gnu.announce&selm=mailman.1865.1091019304.1960.info-gnu%40gnu.org)
- The GNU Bash Reference Manual (http://www.network-theory.co.uk/bash/manual/), (HTML version (http://www.network-theory.co.uk/docs/bashref/)) by Chet Ramey and Brian Fox. ISBN 0954161777

Bash guides from the Linux Documentation Project:

- Bash Guide for Beginners (http://www.tldp.org/LDP/Bash-Beginners-Guide/html/)
- BASH Programming - Introduction HOW-TO (http://www.tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html)
- Advanced Bash-Scripting Guide (http://www.tldp.org/LDP/abs/html/)

Other guides and tutorials:

- Beginners Bash Tutorial (http://hypexr.homelinux.org/bash_tutorial.html)
- Advancing in the Bash Shell tutorial (http://deadman.org/bash.html)
- Linux Shell Scripting Tutorial - A Beginner's handbook (http://www.cyberciti.biz/nixcraft/linux/docs/uniqlinuxfeatures/lsst/)

Retrieved from "http://en.wikipedia.org/wiki/Bash"

Categories: Domain-specific programming languages | Scripting languages | Unix programs/Shells | GNU project software